Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2014, Article ID 124701, 14 pages
http://dx.doi.org/10.1155/2014/124701

Research Article

Recovering Software Design from Interviews Using the NFR Approach: An Experience Report

Nary Subramanian,[1] Steven Drager,[2] and William McKeever[2]

[1] *Department of Computer Science, University of Texas at Tyler, Tyler, TX 75799, USA*
[2] *Information Directorate, Air Force Research Lab, Rome, NY 13441, USA*

Correspondence should be addressed to Nary Subramanian; nsubramanian@uttyler.edu

Received 30 June 2013; Revised 11 January 2014; Accepted 19 January 2014; Published 17 April 2014

Academic Editor: Gerardo Canfora

In the US Air Force there exist several systems for which design documentation does not exist. Chief reasons for this lack of system documentation include software having been developed several decades ago, natural evolution of software, and software existing mostly in its binary versions. However, the systems are still being used and the US Air Force would like to know the actual designs for the systems so that they may be reengineered for future requirements. Any knowledge of such systems lies mostly with its users and managers. A project was commissioned to recover designs for such systems based on knowledge of systems obtained from stakeholders by interviewing them. In this paper we describe our application of the NFR Approach, where NFR stands for Nonfunctional Requirements, to recover software design of a middleware system used by the Air Force called the Phoenix system. In our project we interviewed stakeholders of the Phoenix system, applied the NFR Approach to recover design artifacts, and validated the artifacts with the design engineers of the Phoenix system. Our study indicated that there was a high correlation between the recovered design and the actual design of the Phoenix system.

1. Introduction

Design recovery in software systems involves obtaining design from artifacts such as code, system documentation, and execution environment, with the primary objectives being reduced maintenance costs, system enhancement, or system reengineering [1–3]. Design recovery is especially important for legacy systems where only a few software artifacts exist to aid their understanding: for example, there are systems that are decades old and will serve useful purposes if reengineered but whose only artifacts are limited documentation, a few stakeholders willing to share their experiences with the system, and the executable binary code [4].

Usual techniques for design recovery include code-based and domain knowledge-based techniques. Code-based techniques [4, 5] start with parsing the code and identifying elements and then obtain the design. In doing so an intermediate representation of the code is derived such as the hammock graphs, dependency graphs, or control flow graphs. However, as mentioned in [4], knowledge about architecture, design decisions, and design constraints cannot be fully obtained from code analysis alone. Domain knowledge-based techniques are used for program comprehension [6] that uses knowledge representation such as graphs and trees and apply classical reasoning techniques to retrieve design patterns; they are mostly used to augment code-based techniques to capture higher level abstraction of designs; but this again does not fully capture the high level architecture for the system [7].

In our project, the objective was to obtain designs by interviewing stakeholders of a system. Stakeholders included users and managers. We were provided a rough idea of the environment in which the system operates but no access to system documentation. We were given time slots to interview system stakeholders and we were free to ask questions relevant to the system. From these interview notes we were to recover the design of the system. For the purpose of validation we were asked to recover design for the Phoenix system [8], which is a middleware system used by the US Air Force. The Phoenix system is a middleware system developed

by the Air Force for use in tactical and enterprise systems. Phoenix belongs to the class of message-oriented middleware and provides transparent message transport facilities over multiple operating systems and network protocols. Applications can communicate with Phoenix using well-defined interfaces and exchange real time data. Phoenix operates using producer-consumer paradigm and uses store-and-forward technique for increased reliability of message transport over any unreliable communication links. The code base of Phoenix is over 100 K lines of code and consists of fifteen subsystems. Phoenix was a suitable candidate system for reverse engineering because of the ready availability of its user base as well as the access to its developers for validation. Additionally, Phoenix system's complexity is similar to those legacy systems of which Air Force is interested in recovering the designs.

Our process of reverse engineering from stakeholder interviews was guided by the NFR Approach [9–13], where NFR stands for nonfunctional requirements. The NFR Approach is knowledge driven, permits design trade-off analysis, provides a systematic process for recovering designs from domain knowledge, and rationalizes the process of design recovery from domain knowledge. It is a goal-based approach that has been used for forward engineering software systems where, starting with requirements for the system, the designs were developed [11, 13, 14]; however, since it can also capture information from documents [15], we find it suitable for design recovery from documentary evidence obtained from interviewing users.

During design recovery we obtain multiple views for the architecture [16–18] including component and connector view, detailed structural view, logical deployment view, physical deployment view, and use-case scenarios. In this NFR Approach-based process, we attempted to obtain several of these artifacts for the Phoenix system. The developers of Phoenix validated the resulting designs.

This paper is organized as follows: Section 2 discusses documentation of stakeholder interviews, Section 3 describes the NFR Approach briefly, Section 4 applies the NFR Approach to recover design for the Phoenix system, Section 5 discusses validation and lessons learnt, and Section 6 concludes the paper and provides directions for future work.

1.1. Related Work. Existing techniques for reverse engineering primarily rely on source code as a basis for design and requirement recovery [4, 7]. They can be categorized into static and dynamic approaches and into automatic and semi-automatic approaches. Static approaches reverse engineer from source code by creating an intermediate graphical representation of the code [5, 19–21] while dynamic approaches actually execute the code [22–24]. Automatic approaches [22, 25, 26] are completely tool-based that recover designs and requirements from source code, while semiautomatic techniques employ manual intervention in the process [22, 27, 28]. The NFR Approach has also been used in static source code-based reverse engineering case studies [20] with encouraging results.

There have been attempts to reverse engineer systems from domain information. An inductive learning approach to recovering specifications has been described in [29] where the program behavior is studied. An approach that uses runtime use-case scenarios to extract requirements can be seen in [24]. An approach that uses financial information together with code details in C programming language is given in [28]. Also, design documentation and other system documentations for recovering ADA code are described in [1]. Design recovery by employing Unified Modeling Language to extract class diagrams has been shown in [21, 23], while the Unified Process has been used to recover design elements in [30] by employing use-case scenarios.

There have also been attempts to recover design information using metrics [31] such as weighted number of methods per class and coupling data to associate programs or subroutines to objects. They created a call graph structure to analyze data-intensive COBOL programs and classify them. A clustering technique that groups components realizing specific functions of a web application [32] has been proposed to develop UML diagrams for a web application.

The NFR Approach described in this paper appears to be unique in aiming to capture designs from stakeholder experiences and explicitly including the human feedback as a basis for design recovery. A high-level presentation of our approach was presented in [33]—this paper is a significantly more detailed description of our work.

2. Interviewing Stakeholders

In order to interview stakeholders we needed to first identify them. Phoenix development team had a list of users of Phoenix from which we shortlisted three major system development projects within the Air Force: Marti, Command and Control (C2), and E-Phoenix. Each of these projects develops a sophisticated system using Phoenix to provide communication linkages between system components. We were given contact information for users in each of these projects who were experienced in using the Phoenix system. Also the development team's manager was interviewed. Questions asked during interviews included the following.

(Q1) List problems faced before Phoenix was developed.

(Q2) Before Phoenix, how were the business processes it automated performed?

(Q3) List the main objectives for Phoenix system.

(Q4) List the business processes that Phoenix satisfies.

(Q5) List three problems with Phoenix system.

Each of these questions served a specific purpose. Q1 helped understand the problems that caused migration to Phoenix; Q2 was aimed at capturing the business domain knowledge; Q3 helped understand the expectations for the Phoenix system; Q4 helps understand how Phoenix was used for automating processes and obtain an understanding of the capability of Phoenix; Q5 lists problems that stakeholders now have with Phoenix itself which help understand unmet expectations. The interview was conducted in an informal but structured manner. The opinions of stakeholders were recorded and any misunderstandings were clarified

(1) *Why do you use Phoenix?*
Marti is a system of databases and messages to get images from aircraft to ground. The underlying infrastructure is Phoenix and it controls the dataflow. Data are typically images and CoT (cursor on target) messages; CoT messages are both text and XML. Phoenix usage is completely transparent; (user) did not know that Phoenix was being used until the code was reviewed.

(2) *Before Phoenix was available how did you accomplish these functions?*
Marti was never used without Phoenix

(3) *How do you use Phoenix? Can you list the steps?*
From a user perspective:
(a) On the Falcon View (a PC based mapping application) make a query for an image;
(b) Images are retrieved from Phoenix;
(c) Get the images.
(d) Click to see them on Falcon View
(e) View images or use RouteScout

From the Phoenix perspective:
(a) Add images or CoT messages to Postgres database
(b) No need to specific destination (ports are defined—it is a closed set)
(c) Images/messages sent to a fixed destination

(4) *List problems with Phoenix?*
No user documentation for Phoenix—a DFD of process to use Phoenix will be useful (especially if it includes data formats).
For Marti even the developer documentation is not there.
The Marti system is dependent on Postgres database—if Postgres fails then whole system fails.

Box 1: Answers obtained by interviewing a user of the Phoenix system.

by seeking subsequent feedback. This was an iterative and incremental process of knowledge gathering, where one set of interviews led to questions for other interviews.

An example response is given in Box 1 from one of the users. The interviews were conducted in an environment friendly to the interviewees, either in their office or in a conference room, for no more than thirty minutes each time. During interviews questions were asked to clarify responses. Clarifications were also asked by follow-up meetings or by e-mail. Responses during interviews were recorded by hand and subsequently transcribed using word processing software.

We then applied the NFR Approach to identify design alternatives for the Phoenix system from user responses. The use of the NFR Approach for this purpose is discussed in the next two sections.

3. The NFR Approach

The NFR Approach is a goal-oriented approach that can be applied to determine the extent to which objectives are achieved by a process or product. NFR stands for nonfunctional requirements, which represent properties of a system such as reliability, maintainability, and flexibility and could equally well represent functional objectives and constraints for a system. In this paper we applied the NFR Approach to reversely engineer a software system by evaluating whether a specific design element satisfied specific requirements for the system. The NFR Approach also allows functional requirements to be represented as hardgoals [34]. The NFR Approach uses a well-defined ontology for this purpose that includes NFR softgoals, hardgoals operationalizing softgoals, claim softgoals, contributions, labels, and propagation rules; each of these elements is described briefly below (details may be seen in [9]). Furthermore, the NFR Approach uses the concept of satisficing, a term borrowed from economics, which indicates satisfaction within limits instead of absolute satisfaction, since absolute satisfaction of NFRs is usually difficult.

NFR softgoals represent NFRs and their decompositions. Elements that have physical equivalents (process or product elements) are represented by operationalizing softgoals and their decompositions. During decompositions (of either the NFR softgoals or the operationalizing softgoals), AND decomposition is used when each child softgoal of the decomposition has to be satisficed for the parent softgoal to be satisficed but the denial of even one child is sufficient to deny the parent, OR decomposition is used when satisficing of even one child satisfices the parent but all children need to be denied for the parent to be denied, and EQUAL decomposition has only one child for a parent and propagates the satisficing or the denial of the child to the parent.

Hardgoals represent functional requirements and their decompositions. Again hardgoals can be decomposed using AND, OR, or EQUAL decompositions. Contributions (MAKE, HELP, HURT, and BREAK) are made by operationalizing softgoals to the NFR softgoals and hardgoals. Reasons for contributions are captured by claim softgoals, and claim softgoals may form a chain of evidence where one claim satisfies another, which satisfies another, and so on. Each of the four types of contributions has a specific semantic significance: MAKE contribution refers to a strong positive degree of satisficing the objectives (represented by NFR softgoals) by artifacts (represented by operationalizing softgoals) under consideration, HELP contribution refers to a positive degree of satisficing, HURT

contribution refers to a negative degree of satisficing, and BREAK contribution refers to a strong negative degree of satisficing.

Due to these contributions, some of the softgoals acquire labels that capture the extent to which a softgoal or hardgoal is satisficed: satisficed, weakly satisficed, weakly denied (or weakly not satisficed), denied (or not satisficed), or unknown (indicated by an absence of any label attribute). Labels are ranked in satisficing in the order: satisficed > weakly satisficed > unknown > weakly denied > denied. Moreover, high priority softgoals, hardgoals, decompositions, and contributions may be indicated using the criticality symbol.

Propagation rules propagate labels from child softgoal to the parent across decompositions, from operationalizing softgoals to NFR softgoals (or hardgoals) across contributions, and from claim softgoals to contributions; propagation rules aid in the rationalization process of the NFR Approach. Example propagation rules are as follows (details can be seen in [9]).

(R1) Determine labels for all operationalizing softgoals, claim softgoals, and contributions: each is either satisficed, denied, weakly satisficed, weakly denied, or unknown.

(R2) If a softgoal label is satisficed (denied) and it has a MAKE contribution to its parent, then the softgoal propagates its label to the parent.

(R3) If a softgoal label is satisficed (denied) and it has a BREAK contribution to its parent, then the softgoal propagates denied (satisficing) label to its parent.

(R4) If all labels propagated to a parent (either softgoal, hardgoal, or a contribution) are satisficed (denied), then that parent is satisficed (denied).

(R5) If there is a mix of labels propagated to a parent (either softgoal, hardgoal, or a contribution) and then if most of the labels are satisficed (denied), then the parent is weakly satisficed (weakly denied).

(R6) If the label of a softgoal is unknown, then if it is involved in an AND contribution, its label is assumed to be satisficed and if it is involved in an OR contribution, its label is denied.

(R7) In the case of AND-decomposed softgoals, if even one child softgoal has a denied label, then the parent is denied; otherwise the parent is satisficed.

(R8) In the case of OR-decomposed softgoals, if even one child softgoal has a satisficed label, then the parent is satisficed; otherwise the parent is denied.

(R9) If a contribution is denied, then a MAKE contribution becomes a BREAK contribution and vice versa; a weakly satisficed contribution becomes a weakly denied contribution and vice versa.

The propagation rule R1 states that a softgoal can have one of five labels—satisficed, weakly satisficed, weakly denied, denied, and unknown; this is true for hardgoals as well. Rules R2 and R3 state the label propagated by a softgoal to its parent via MAKE or BREAK contributions. Rules R4 and R5 state the labels for parents based on contributions from their children. Rule R6 states what to do when a softgoal label is of unknown type—in this rule we have assumed an open policy approach wherein ignorance is considered favorably from a satisficing viewpoint: if a softgoal label is unknown, its impact on satisficing its parent is negligible—we let impact from softgoals whose satisficing is known to dominate the labels propagated to the parent. We discuss this further in a later section. Rules R7 and R8 state the labels propagated to the parent softgoal involved in an AND or OR decomposition with its children. Rule R9 states how unsatisficed contributions are treated: if a contribution is unsatisficed, then its type changes to the opposite type of satisficing.

These elements of the NFR Approach are captured in a graphical representation called the Softgoal Interdependency Graph (SIG). Each softgoal is named using the convention *Type* [*Topic*] where *Type* is the name of the softgoal and *Topic* is the context where the softgoal is used; *Topic* is optional for a softgoal and for a claim softgoal; the name may be the justification itself.

We applied the NFR Approach to analyze stakeholder experiences by transforming interview notes into SIGs for further analysis—this process is discussed in the next section. The partial ontology of the NFR Approach is shown in Figure 1.

3.1. Process for Applying the NFR Approach for Design Recovery. The steps for applying the NFR Approach for design recovery from interview notes are as follows.

(1) For each interview note develop the SIG including hardgoals, NFR softgoals, operationalizing softgoals, and claim softgoals.

(2) Apply satisficing labels and propagate labels up the SIG using the propagation rules.

(3) Identify those NFR softgoals that are satisficed by the Phoenix system.

(4) Use catalogs to identify those design alternatives that satisfice the NFR softgoals identified in step (3); these design alternatives are represented as operationalizing softgoals.

(5) Develop contributions to each operationalizing softgoal identified in step (4) from other SIGs obtained in step (1).

(6) Propagate labels from step (2) to each operationalizing softgoal in step (4). The design alternative (operationalizing softgoal) with the most satisficing label is the best candidate.

In the first step, develop the SIG that includes hardgoals and softgoals for each interview note. Then apply satisficing labels (satisficed, weakly satisficed, denied, and weakly denied) to all goals in a SIG and determine the satisficing of hardgoals and softgoals. Since design alternatives are compared along nonfunctional requirements when each alternative satisfies functional requirements [35], we identify those NFR softgoals that have been satisficed. We then

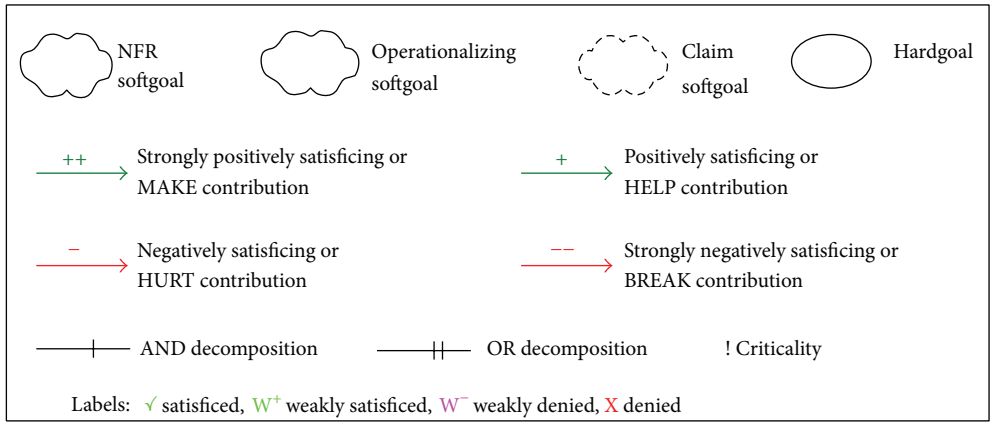

Figure 1: Partial ontology of the NFR Approach.

use catalogs to identify design alternatives that satisfice the identified NFR softgoals—these design alternatives become operationalizing softgoals in the SIG. We obtain corroboration from other SIGs for each of the design alternatives by identifying contributions between these SIGs and the design elements represented as operationalizing softgoals. We propagate labels to these operationalizing softgoals using the propagation rules of the NFR Approach. The design alternative with most highly ranked label is the best design candidate.

At this time we would like to contrast the NFR Approach with some of the other techniques in literature. The concept analysis technique [36] develops a tree of concepts and subconcepts that can be used to modularize the code for a system—concepts can represent functions for the system and serve as the basis for design recovery; however, the NFR Approach uses design catalogs that are justified by stakeholder claims using a rationalization process for design recovery. Also, the UML [37] while providing stereotypes for capturing design information does not have constructs for decompositions, contributions, labels, and propagation rules.

4. Application of the NFR Approach for Recovering Architecture of Phoenix System

The first step in applying NFR Approach is to generate the SIGs for each user interview document. The SIG obtained from the interview note of Box 1 is shown in Figure 2. At the bottom of the figure is the operationalizing softgoal, Phoenix user [Marti], which represents the fact that this user uses Phoenix for the Marti system developed by the Air Force. At the top are the hardgoals and softgoals extracted from the interview note. There are two NFR softgoals: Transparency [Users] and Reliability [Database], which refer to the fact that, respectively, users of Phoenix find it transparent and that the database is reliable. These two NFR softgoals arise from the two statements in the interview:

> "Phoenix usage is completely transparent." and
>
> "…system dependent on Postgres database (of Phoenix)—if Postgres fails then whole system fails."

There are also three hardgoals that represent the functional requirements of the system and arise from user statements: Controls Dataflow; Transports Images, Text, and XML Data; Uses Postgres Database.

The arrows between operationalizing softgoal and hardgoals/NFR softgoals are the contributions made by the operationalizing softgoal to the hardgoals and NFR softgoals. As per the user, the Phoenix system satisfices its hardgoals and so MAKE contributions exist to all hardgoals. Justifications for these MAKE contributions are given by the claim softgoals, which refer to the statement in the interview note that justifies the contributions. Likewise, Phoenix has a MAKE contribution to the NFR softgoal Transparency [Users] but has a BREAK contribution to the NFR softgoal Reliability [Database], and justifications for these are captured by the claim softgoals, which refer to the appropriate user statements.

We now apply the propagation rules to the SIG. All claim softgoals are satisficed since they are statements made by the user. Since all contributions between claim softgoals and the contributions they justify are MAKE contributions, by rule R2, all contributions between operationalizing softgoal Phoenix and hardgoals/NFR softgoals are all satisficed. Again by rule R2, all hardgoals are satisficed. Likewise, by rule R2 NFR softgoal Transparency [Users] is satisficed while by rule R3, the NFR softgoal Reliability [Database] is denied. This also means the SIG transcribes the interview note correctly since conclusions from the SIG match those on the note. This completes steps 1 and 2 of the process given in Section 3.1.

As can be seen from the SIG of Figure 2, the only satisficed NFR softgoal is Transparency [Users]. In the next step we turn to catalogs to identify architectural styles that will help achieve transparency for users: three techniques stand out—domain-specific (DS) middleware [38], service-oriented architecture (SOA) using web services [39], and proprietary SOA [40]. DS middleware is useful in a specific domain only; it will need significant reconfiguration for use in another domain. SOA is based on a structure of loosely coupled services interacting over a common bus; Web Services (a specific case of SOA) support brokering of loosely coupled services based on SOAP (Simple Object Access Protocol) protocol and UDDI (Universal Description,

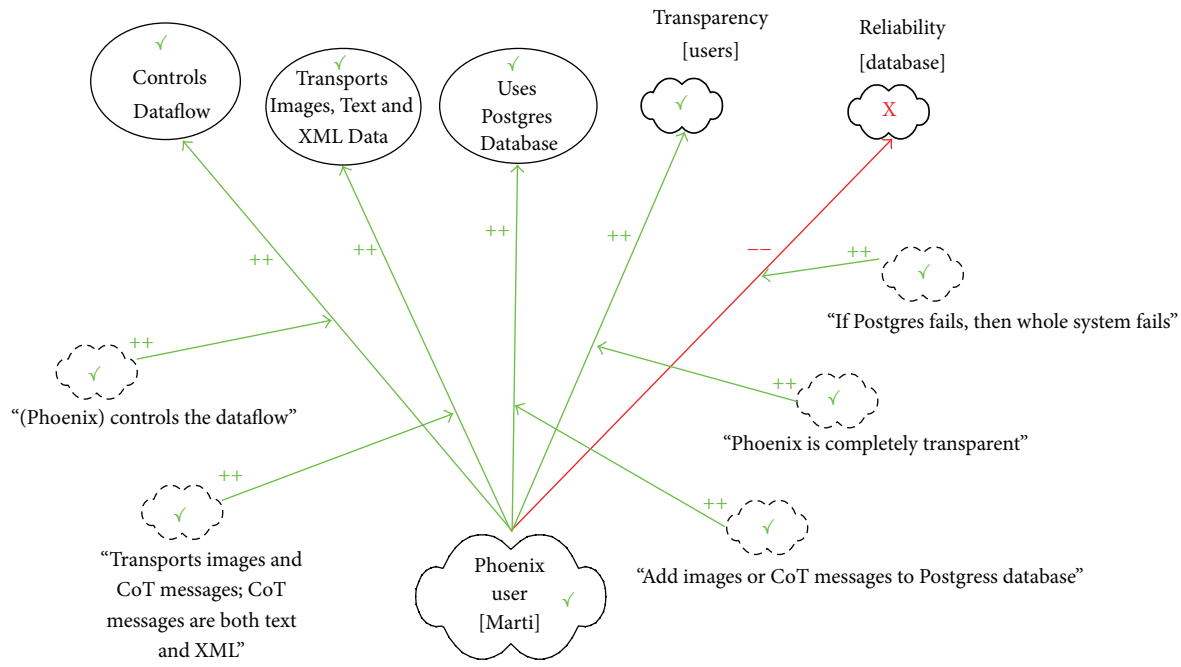

Figure 2: SIG representation of the interview note of Box 1.

Discovery, and Integration). Proprietary SOA uses its own protocols and underlying artifacts (including hardware communication channels) to seamlessly transport data.

The SIG of Figure 3 analyzes these architectural alternatives to identify the most suitable candidate for the Phoenix system. At the bottom of this figure is the SIG of Figure 2. In the middle are three operationalizing softgoals representing the three alternatives. At the top is the SIG from interviewing other users (from C2 and E-Phoenix projects)—only the relevant portions from these two SIGs are shown in Figure 3 and they are shown inverted for purposes of understanding. These corroborating users identified four hardgoals: services need channels, provides services, provides SOAP interaction, and provides REST services (REST standing for Representational State Transfer [41]). Of these hardgoals, only two are satisficed based on user statements; the remaining two were on the wish lists of users but not satisficed by Phoenix. The next step is to identify which of the alternatives satisfies all the functional requirements represented by hardgoals. For this purpose, contributions between hardgoals and these alternatives are drawn and claim softgoals capture justifications for these contributions. These justifications are shown in Table 1. There are also contributions between the three hardgoals in the lower SIG (it should be noted that Figure 3 is one SIG—however, for purposes of explanation we are dividing it into lower SIG that repeats Figure 2 and an upper SIG that captures corroborating statements from other users): Controls Dataflow; Transports Images, Text, and XML Data; Uses Postgres Database. Contributions from these hardgoals to the alternatives are all MAKE except for the one BREAK contribution from Uses Postgres Database hardgoal to the SOA [Web Services] operationalizing softgoal—to avoid clutter (and, more importantly, since they do not have negative impact evaluation using propagation rules) only this BREAK contribution is shown in Figure 3.

Table 1: Claims in Figure 4.

Claim softgoal	Justification
C1	Domain-specific middleware does not support multidomain services.
C2	SOA using web services does not provide channels (which are hardware dependent communication primitives).
C3	Proprietary SOA can be developed with required channels.
C4	Web services provide SOAP interaction.
C5	Proprietary SOA can be developed to provide multidomain services.
C6	Web services can provide REST services.
C7	Web services need not use Postgres database.

We now apply propagation rules to find which alternative is most appropriate. Let us consider the labels received by each alternative—these are shown in Tables 2, 3, and 4. All claims C1 through C7 in Table 1 are satisficed and they have MAKE contributions to their parent contributions—therefore, by rule R2, all parent contributions in Figure 3 are satisficed, which means none of the contributions change their nature (i.e., rule R9 does not apply for any contribution).

The final label of the operationalizing softgoal DS middleware will be, by rule R5, weakly satisficed since most of the labels propagated to this softgoal are satisficed. This satisficing is indicated in Figure 3.

The final label of the operationalizing softgoal SOA [Web Services] will be, by rule R5, weakly denied since most of the

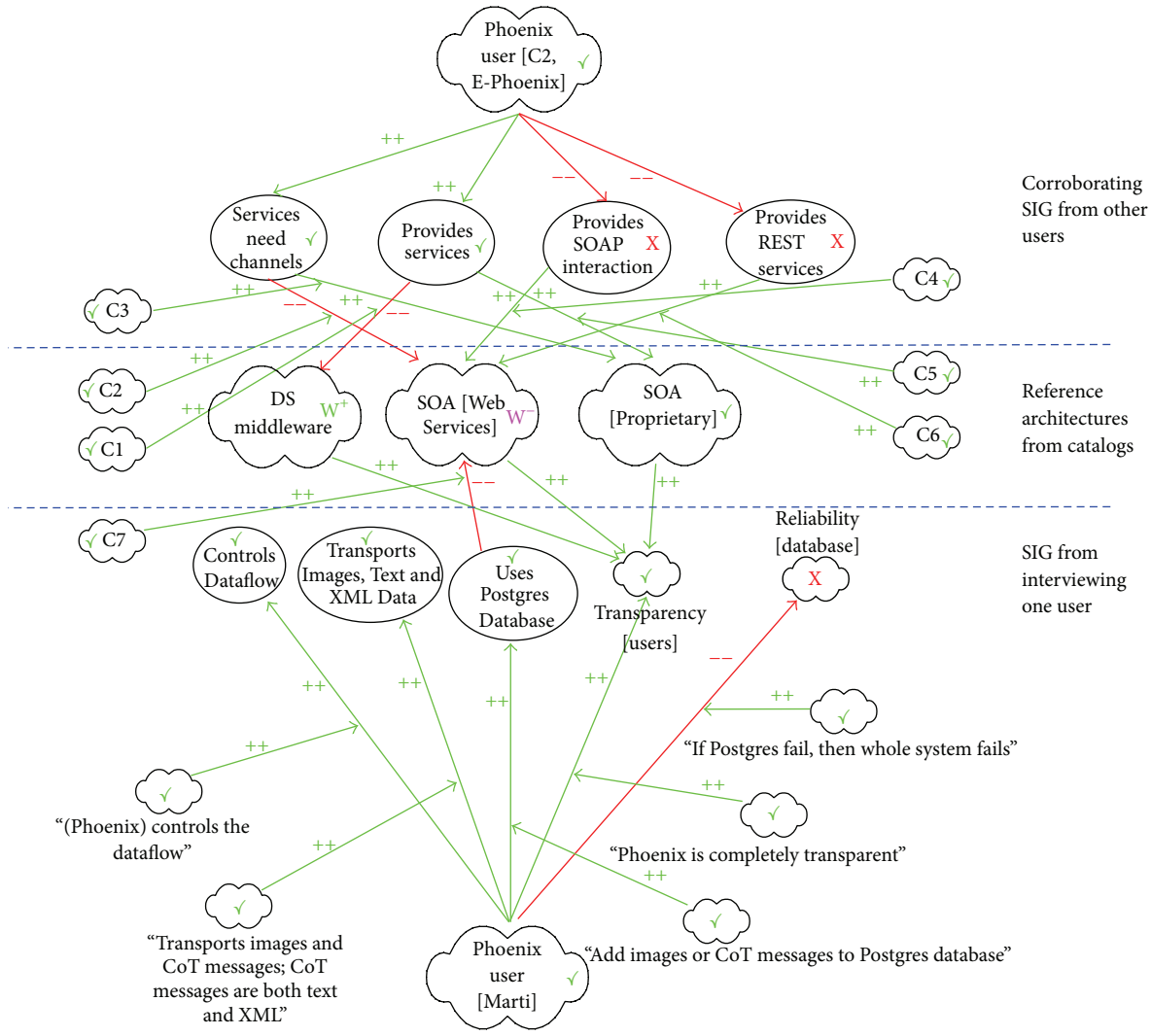

Figure 3: SIG for recovering architecture of Phoenix system.

Table 2: Labels propagated to the domain specific middleware alternative.

Source hardgoal	Hardgoal label	Hardgoal contribution	Label propagated	Rule applied
Provides Services	Satisficed	BREAK	Denied	R3
Controls Dataflow	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Transports Images, Text, and XML Data	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Uses Postgres Database	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2

labels propagated to this softgoal are denied. This satisficing is indicated in Figure 3.

The final label of the operationalizing softgoal SOA [Proprietary] will be, by rule R5, satisficed since all the labels propagated to this softgoal are satisficed. This satisficing is indicated in Figure 3.

Since SOA [Proprietary] is the satisficed operationalizing softgoal, the most appropriate architecture for Phoenix is a proprietary SOA architecture. Therefore, the initial architecture for the Phoenix system is shown in Figure 4.

In Figure 4, the SOA infrastructure component has the services interpretation module, execution module, message passing module, and the data storage module. The exposed services form the set of services that clients of the system can invoke. However, we need to flesh out details of this architecture. There are references to databases in interviews

Table 3: Labels propagated to the web services alternative.

Source hardgoal	Hardgoal label	Hardgoal contribution	Label propagated	Rule applied
Services need channels	Satisficed	BREAK	Denied	R3
Provides SOAP interaction	Denied	MAKE	Denied	R2
Provides REST services	Denied	MAKE	Denied	R2
Controls Dataflow	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Transports Images, Text, and XML Data	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Uses Postgres Database	Satisficed	BREAK	Denied	R3

Table 4: Labels propagated to the proprietary SOA alternative.

Source hardgoal	Hardgoal label	Hardgoal contribution	Label propagated	Rule applied
Services need channels	Satisficed	MAKE	Satisficed	R2
Provides services	Satisficed	MAKE	Satisficed	R2
Controls Dataflow	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Transports Images, Text, and XML Data	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2
Uses Postgres Database	Satisficed	MAKE (not shown in Figure 4)	Satisficed	R2

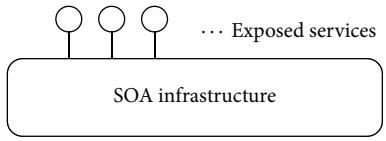

Figure 4: Initial architecture for the Phoenix system.

but these need not necessarily refer to databases inside of Phoenix. For the next level of design refinement, we will apply the NFR Approach to available public documentation of the Phoenix system.

4.1. Refining the Initial Architecture. There are two documents available for public view: the user manual (called the Final Technical Report or FTR) and a published paper [8]. Using information from these two documents we create a SIG for architecture refinement as shown in Figure 5. At the bottom of this figure is the SIG from documentation. The operationalizing softgoal Phoenix [Documentation] is AND-decomposed into softgoals FTR and Paper [Combs] referring to the two documentation sources. Some of the statements from this documentation are represented as hardgoals, and contributions are shown from the operationalizing softgoals to these hardgoals. All contributions are MAKE and justifications for these are the corresponding statements in the documents (these claim softgoals are not shown to avoid clutter). At the top of the SIG of Figure 5 are three hardgoals obtained from users of Marti and E-Phoenix systems and these hardgoals all receive MAKE contributions as well. In the middle are SOA elements from catalogs such as [39]. SOA needs mediation facility; a mediation facility stores register information of publishers so that when consumers request, they are connected with appropriate publishers by the mediation facility. SOA provides services and based on the hardgoals the services are Query Service (QS) that responds to queries from Inquisitors (agents that query), Dissemination Service (DS) that provides information to consumers, Subscription Service (SS) that producers of information subscribe to, Service Brokering Service (SBS) that all services register with, and Repository Service (RS) that controls datastores. The proprietary SOA alternative has channels between services, a Registry that helps SBS register services, and Datastores based on Postgres for storing information. These elements of SOA are operationalizing softgoals. Also it can be seen that the operationalizing softgoal services are AND-decomposed into five child softgoals representing the five services discussed above. Moreover, we know that mediation facility in Phoenix seems to be provided by SBS (Service Brokering Service) since all services need to register with it—the MAKE contribution between hardgoal services register with SBS and operationalizing softgoal mediation facility captures this. Since SBS registers services, the Registry is associated with it (another MAKE contribution indicates this relationship). We know from interviewing one of the users that a channel needs to exist between each pair of services for communication—therefore MAKE contribution to operationalizing softgoal channels. We also know from FTR that Phoenix provides ten services and their names are known—these form the services provided by the SOA architecture. We also know that Inquisitor is connected to QS and DS, Consumer to DS, and Producer to SS. We also know that RS controls four types of repositories through

Figure 5: Architecture refinement using the NFR Approach.

a repository interface (from FTR). By repeated application of propagation rule R2, all operationalizing softgoals representing SOA elements are satisficed. We then put all this information together to get the refined logical architecture of Figure 6.

The architecture of Figure 6 is the component and connector view of Phoenix system [18]. All components in circles indicate services, the rectangular block in the middle is the mediation facility (SBS), and all drum-shaped figures represent datastores and registry. All datastores are Postgres databases. Channels between services are represented by green arrows and normal arrows represent interfaces to databases.

4.2. Discussion of Design Documents Recovered. Using a similar approach of extracting information from documents, we recovered the following design elements for the Phoenix system:

(1) activity diagrams for different services;

(2) detailed component and connector views;

(3) deployment models.

Figures 7, 8, 9, and 10 show a few of the design views recovered for the Phoenix system. Figure 7 shows the activity diagram for a producer in the Phoenix system: initially all actors register their services with SBS (Service Brokering Service); when a producer has information to send, it establishes a channel with SS (Submission Service) and sends information to it. When a subscriber needs this information, it gets it from SS via the SBS and a copy is also stored in the RS (Repository Service).

The logical deployment diagram is shown in Figure 8 where there are seventeen services (the ten basic services and seven overhead services for Phoenix system such as the channel maintenance services) inside the deployed system. The physical deployment diagram of Figure 9 shows that the seventeen services are deployed as jar files inside the

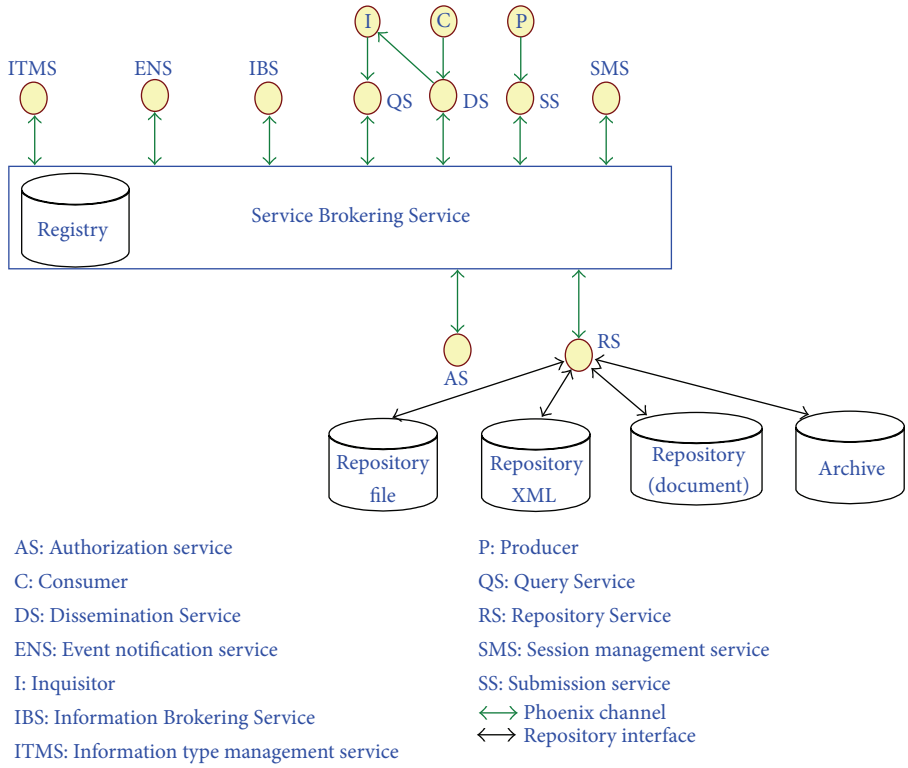

Figure 6: Refined logical architecture for Phoenix.

SpringSource Application Server [42], which resides on top of the Java Virtual Machine (JVM) which in turn runs on top of either Windows or Linux operating systems.

Figure 10 shows the detailed view of one service, the Submission Service (SS). The SS has four major components: Input Channel Manager, Information Validator, Policy Manager, and Forwarder. Inputs are received by the Input Channel Manager, which is processed by the Input Processor. Events (such as informing SBS) are fired by the Event Firer and messages are acknowledged by the Acknowledger. The Information Validator component has four subcomponents: ITMS Communicator, Cache Manager, Validator, and Transactions Logger. ITMS Communicator sends the input to the Information Type Management Service (ITMS) for confirming if the type of input is acceptable; the Cache Manager caches the input for later replay in case of errors; the Validator confirms that the format of the input is valid; the Transactions Logger logs the transactions for later audit. The Policy Manager component guides the functioning of the remaining three components by ensuring that established policies are followed in processing messages; the Forwarder component relays the processed message downstream.

5. Validation and Lessons Learnt

Recovered designs were validated by three of the original Phoenix development engineers. We presented the designs along with a questionnaire that required them to evaluate the artifacts by comparing them with the system documents for Phoenix (which we did not have access to) in the spirit of the Delphi technique [43]. The developers were provided activity diagrams, multiple views of architecture, and logical and physical deployment models. Based on the developers' responses the results are tabulated in Table 5.

As can be seen in Table 5, the component and connector view were 90% correct (in terms of the number and types of components and connections); however, there seems to be no central broker in the implementation of Phoenix and bidirectionality of links is an incorrect assumption (there are separate links for forward and reverse connections) and more data analysis in terms of further claim softgoal rationalizations can help uncover this information. The activity diagram for the scenario of producer submitting information to the dissemination service is 60% correct in terms of activities and 88% correct in terms of links but further iteration of information discovery process would have helped improve accuracy. Physical deployment model and detailed view were mostly correct though further iterations will help improve accuracy.

One of the lessons we learnt was the importance of domain knowledge for improving efficiency of design recovery—the number of iterations needed to get sufficient information from the stakeholders to capture design rationales depended on the complexity of the system. Domain knowledge could appear in the SIG as a claim softgoal or operationalizing softgoal. For example, the knowledge of what Marti system does will help us understand user's viewpoint much better.

In the SIG of Figure 3 we concluded the architecture type based on contributions to one NFR softgoal Transparency

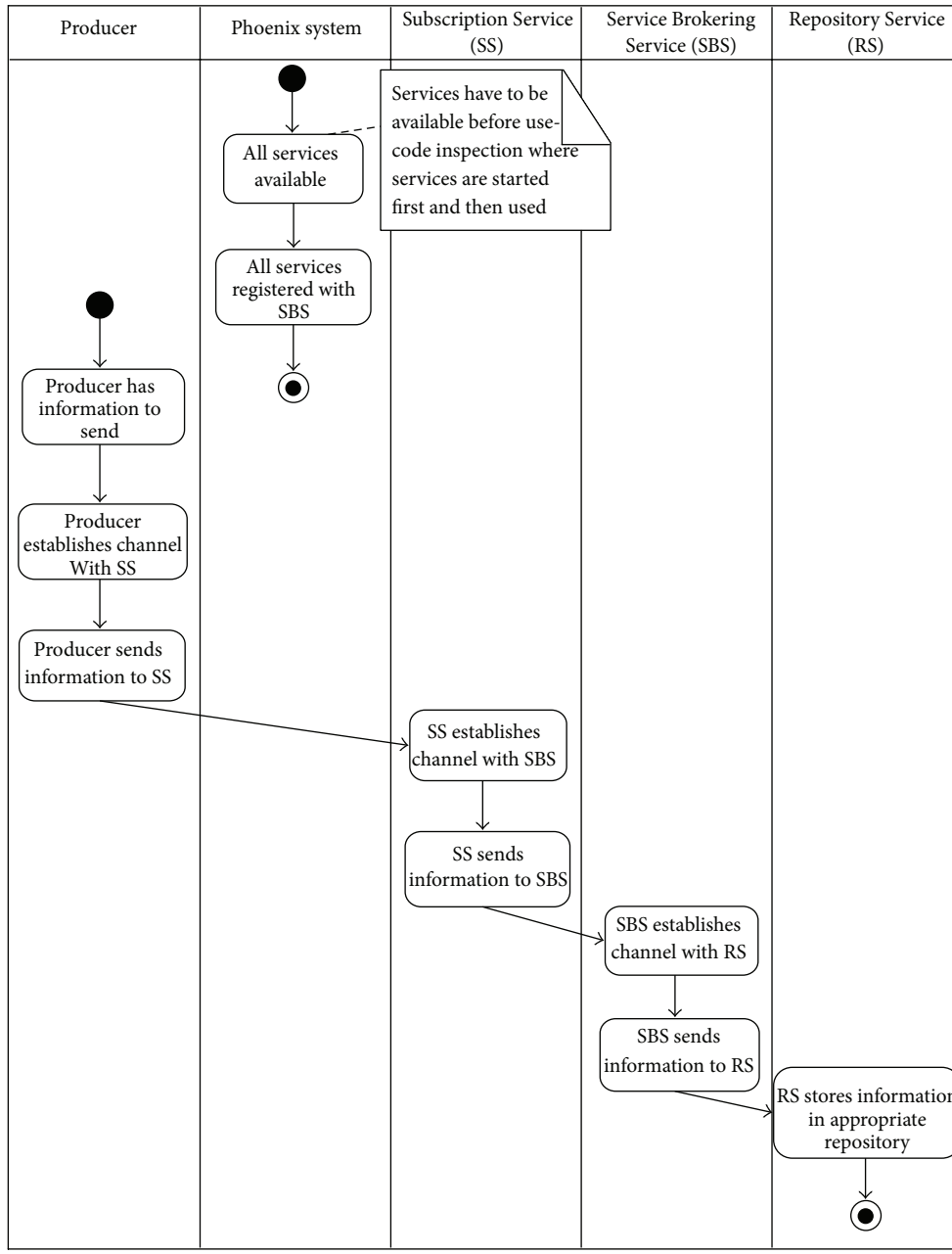

Figure 7: Activity diagram for producer actor recovered using the NFR Approach.

Table 5: Validation results.

Item	Positives	Scope for improvement
Component and connector view	90% correct	No central broker; Information Brokering Service (IBS) is more important than SBS; bidirectionality assumptions are incorrect.
Activity diagrams	60% activities correct; 88% links correct	Some of the links had wrong sources and/or destinations; some of the activities were in the wrong order.
Logical deployment model	100% correct	This has sparse information.
Physical deployment model	90% correct	We concluded SpringSource was used for application server—in fact, Java Services Container does this job.
Detailed views of services	93% correct	Forwarder should be Output Channel Manager.

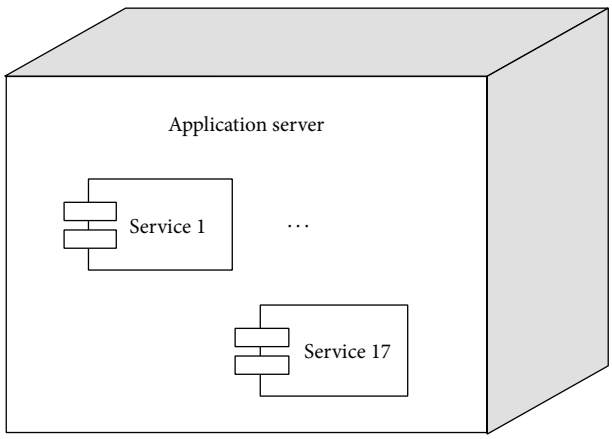

Figure 8: Logical deployment model for the Phoenix system.

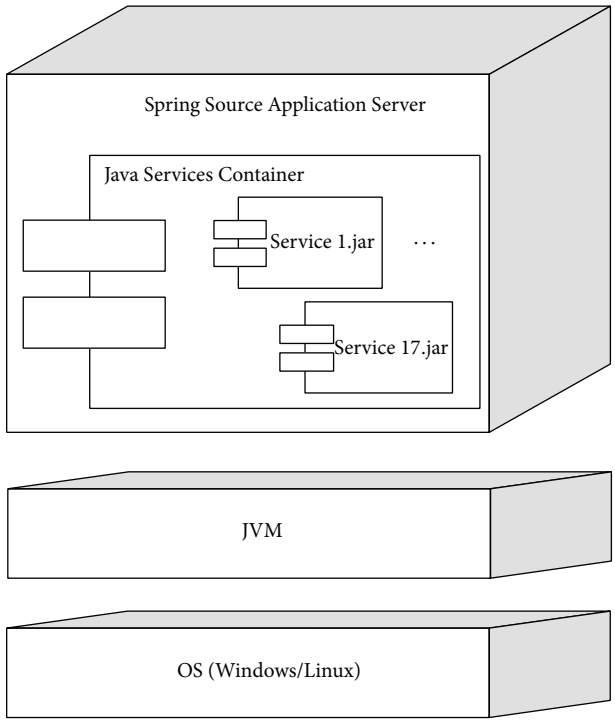

Figure 9: Physical deployment model for the Phoenix system.

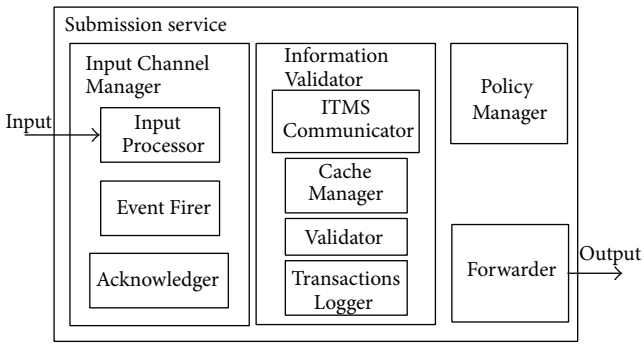

Figure 10: Detailed view for the Submission service of the Phoenix system.

[User]—what happens if more than one NFR softgoal is involved or if contributions are conflicting? In that case, the propagation rules of the NFR Approach can be used for trade-off analysis to determine the suitable candidate. In Figure 3, the current Phoenix architecture does not satisfice Reliability [Database] (conflicting NFRs)—therefore, the three architectural alternatives do not need to be reliable as far as their database is concerned; however, this also provides a design improvement opportunity for the next version of the system being recovered. If another NFR softgoal is involved in a synergistic manner (this is not the case in Figure 3), then architecture alternatives from catalogs will need to satisfice both NFRs to be considered a valid candidate.

A point that needs stressing is that frequent feedback from users will help converge faster for better designs since the feedback can be used to better guide the design recovery process especially when conflicting claims are made by stakeholders—in this case, it is quite possible that knowledge captured by a SIG is incomplete and more details will need to be discovered. Decisions reached during design recovery using the NFR Approach are traceable through the SIG since it helps maintain a historical record—therefore, if a recovered design aspect is found wrong, using SIG, we can trace the source of this fault and determine how to not only correct this fault but also evaluate the impact of this correction on the recovered design. In a sense, the process used by us is both bottom-up and top-down: bottom-up when knowledge from catalogs is employed to help in design recovery and top-down when the process is iterated to obtain finer details from previous iterations.

In our discussion in this paper we did not consider criticalities—if any component of the SIG (hardgoal, softgoal, decomposition, or contribution) is critical, then that component will have to be given special consideration during design recovery. For example, if one of the user requirements was high throughput and users considered this requirement favoring their use of Phoenix, then when selecting the appropriate architecture from catalogs, the one that uses store-and-forward mechanism may not be favored as much as an alternative that uses streaming messages since the latter usually has higher throughput. Likewise, in our discussion we assumed all contributions are satisficed (we did not have to use rule R9); however, in practice, some claims may become invalid as more knowledge is uncovered—in that case we can either deny the claim or deny the contribution or both. Therefore, the view captured by Figure 3 is our current knowledge—if things were to change, they can be easily captured in an updated SIG.

Another point to note is that the hardgoals are actually functional requirements for the Phoenix system; that is, in the process of applying the NFR Approach for recovering design we recover requirements as well-functional requirements in the form of hardgoals and nonfunctional requirements in the form of NFR softgoals. Finally, we used the StarUML tool with softgoal profile module [44] for drawing the SIGs—this tool helped us quickly create graphs and maintain versions of SIGs that served as historical records. This tool also keeps track of all SIG elements so that in spite of clutter this tool helps to quickly propagate labels up the SIG using propagation rules.

Therefore the process of design recovery from stakeholder interviews using the NFR Approach includes three steps. In the first step we identify and interview stakeholders associated with the target software system, collect and study software artifacts associated with the system, and study literature to acquire domain knowledge. In the second step, create SIGs from interviews, obtain candidate designs from catalogs, apply labels and propagation rules, and identify the most appropriate candidate based on user information. In the third step, develop SIGs to obtain design views for the architecture such as component and connector view, activity diagrams, detailed logical diagram, logical deployment diagram, and physical deployment diagram.

6. Conclusion and Future Work

Stakeholder views provide a unique viewpoint into a software system—their perception of the functionality of the software system can help verify whether the system is indeed achieving its objectives. Stakeholders for software systems in production (i.e., working or executing software systems) include users and managers. Moreover, for several legacy systems stakeholder views based on their experiences with the software system may be the only major source of information for the system since the original system documentation may no longer be available [4]. In this paper we discuss how we recovered the design for a software system called Phoenix [8] for the US Air Force from stakeholder interviews. Stakeholders were interviewed, catalogs of design information were created, and the NFR Approach [9–13] was applied to analyze interview notes and recover the designs. Activity diagrams, multiple views of architecture, and logical and physical deployment models were generated for the Phoenix system. We validated the recovered design artifacts by feedback from Phoenix's development engineers. The results are encouraging with many design views matching closely that of the developers' designs, which gives confidence that our process may be reused for other systems. The NFR Approach employed for analysis provides the ability to trade off conflicting stakeholder view points, helps record and resolve ambiguity inherent in knowledge acquisition from stakeholder experiences, helps trace recovered designs to the information sources, and maintains historical records in graphical representations called Softgoal Interdependency Graphs (SIGs).

In the future we plan to apply the NFR Approach to other systems at the Air Force to recover designs. We also plan to partially automate this process by incorporating natural language processing to identify elements of SIGs from interview notes, which should speed up the process considerably. Furthermore, integrating natural language processing with tools available for handling NFR Approach, for example, StarUML [44], will enable automating the processing of knowledge recovery from stakeholders. To increase convergence of designs we hope to include data from black-box tests, including performance tests, in the future. However, we believe that design recovery from stakeholder interviews using the NFR Approach is a promising technique for reverse engineering software systems.

Conflict of Interests

The authors certify that there is no actual or potential conflict of interests in relation to this paper. This paper was assigned the Case no. 88ABW-2014-0345 by the US Air Force and was cleared for publication on the 4th of February, 2014.

Acknowledgments

This research was sponsored by Air Force Research Laboratory/Information Directorate, Rome, NY, USA. In the summer of 2011 authors spent many months on this project and they thank several engineers at the lab for helping with their project including Mark Linderman, James Hanna, Vaughn Combs, James Milligan, Chris Schuck, Tim Blocher, Dawn Nelson, and Mark Mowers. They also thank the reviewers of the original version of this paper for their insightful comments that helped them significantly improve the paper.

References

[1] E. J. Byrne, "Software reverse engineering: a case study," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1349–1364, 1991.

[2] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[3] D. E. Wilkening and K. Littlejohn, "Legacy software reengineering technology," in *Proceedings of the 15th AIAA/IEEE Digital Avionics Systems Conference*, pp. 25–30, October 1996.

[4] H. A. Muller, J. H. Jahnke, D. B. Smith, M. A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 47–60, 2000.

[5] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-based reverse-engineering of design components," in *Proceedings of the 21st International Conference on Software Engineering*, pp. 226–235, May 1999.

[6] T. J. Biggerstaff, "Design recovery for maintenance and reuse," *Computer*, vol. 22, no. 7, pp. 36–49, 1989.

[7] G. Canfora and M. Di Penta, "New frontiers of reverse engineering," in *Proceedings of the Future of Software Engineering Conference (FoSE '07)*, pp. 326–341, May 2007.

[8] V. T. Combs, R. G. Hillman, M. T. Muccio, and R. W. McKeel, "Joint battlespace infosphere: information management within a C2 enterprise," in *Proceedings of the 10th International Command and Control Research and Technology Symposium*, 2005.

[9] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, Boston, Mass, USA, 2000.

[10] A. Vemulapalli and N. Subramanian, "Evaluating consistency between BPEL specifications and functional requirements of complex computing Systems using the NFR approach," in *Proceedings of the 4th International Systems Conference (SysCon '10)*, pp. 153–158, April 2010.

[11] L. Chung and N. Subramanian, "Adaptable architecture generation for embedded systems," *Journal of Systems and Software*, vol. 71, no. 3, pp. 271–295, 2004.

[12] L. Chung and N. Subramanian, "Process-oriented metrics for software architecture adaptability," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pp. 310–311, August 2001.

[13] N. Subramanian and L. Chung, "Software architecture adaptability: an NFR approach," in *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*, pp. 52–61, September 2001.

[14] N. Subramanian, S. Drager, and W. McKeever, "Designing trustworthy software systems using the NFR approach," in *Emerging Trends in ICT Security*, B. Akhgar and H. Arabnia, Eds., pp. 203–225, Elsevier, 2014.

[15] N. Subramanian and L. Chung, "Representing and reasoning about agreements ... more agreeably," *Lus Gentium Journal*, vol. 12, pp. 205–258, 2006.

[16] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[17] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Boston, Mass, USA, 2003.

[18] P. Eeles and P. Cripps, *The Process of Software Architecting*, Addison-Wesley, New Jersey, NJ, USA, 2010.

[19] H. A. Muller, M. A. Orgun, S. R. Tilley, and J. S. Uhi, "A reverse engineering approach to subsystem structure identification," *Journal of Software Maintenance*, vol. 5, no. 4, pp. 181–204, 1993.

[20] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. Do Prado Leite, "Reverse engineering goal models from legacy code," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE '05)*, pp. 363–372, September 2005.

[21] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code*, Springer, New York, NY, USA, 2005.

[22] M. Lanza and S. Ducasse, "Polymetric views: a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.

[23] T. Systä, K. Koskimies, and H. Müller, "Shimba: an environment for reverse engineering Java software systems," *Software*, vol. 31, no. 4, pp. 371–394, 2001.

[24] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Towards employing use-cases and dynamic analysis to comprehend mozilla," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 639–642, September 2005.

[25] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[26] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico, "Architectural layer recovery for software system understanding and evolution," *Software: Practice and Experience*, vol. 40, no. 10, pp. 897–916, 2010.

[27] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo, "Cliche-based environment to support architectural reverse engineering," in *Proceedings of the IEEE Conference on Software Maintenance (ICSM '96)*, pp. 319–328, November 1996.

[28] P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari, "Reverse engineering 4. 7 million lines of code," *Software*, vol. 30, no. 2, pp. 129–150, 2000.

[29] W. W. Cohen, "Recovering software specifications with inductive logic programming," in *Proceedings of the 12th National Conference on Artificial Intelligence*, pp. 142–148, August 1994.

[30] P. Dugerdil, "A reengineering process based on the unified process," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 330–333, September 2006.

[31] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino, "Identifying objects in legacy systems using design metrics," *Journal of Systems and Software*, vol. 44, no. 3, pp. 199–211, 1999.

[32] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. de Carlini, "WARE: a tool for the reverse engineering of web applications," in *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, pp. 241–250, 2002.

[33] N. Subramanian, S. Drager, and W. McKeever, "Engineering a trustworthy software system using the NFR approach," in *Proceedings of the Systems and Software Technology Conference*, Salt Lake City, Utah, April 2012.

[34] L. Chung, S. Supakkul, N. Subramanian et al., "Goal-oriented software architecting," in *Relating Software Requirements and Software Architectures*, pp. 91–110, Springer, 2011.

[35] D. Gross and E. Yu, "From non-functional requirements to design through patterns," *Requirements Engineering*, vol. 6, no. 1, pp. 18–36, 2001.

[36] M. Siff and T. Reps, "Identifying modules via concept analysis," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 749–768, 1999.

[37] "Unified Modeling Language (UML)," http://www.uml.org/.

[38] D. C. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, no. 6, pp. 43–48, 2002.

[39] T. Erl, *SOA Design Patterns*, Prentice Hall, New Jersey, NJ, USA, 2009.

[40] D. Sprott and L. Wilkes, "Understanding Service-Oriented Architecture," 2004, http://msdn.microsoft.com/en-us/library/aa480021.aspx.

[41] M. Elkstein, "Learn REST: A Tutorial," http://rest.elkstein.org/.

[42] "SpringSource," http://spring.io/.

[43] C. Hsu - and B. A. Sandford, "The Delphi technique: making sense of consensus," *Journal of Practical Assessment, Research, and Evaluation*, vol. 12, no. 10, 2007.

[44] "StarUML," http://staruml.sourceforge.net/en/modules.php.